BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS
- BLANK PAGES

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 12/02

A1

(11) International Publication Number: WO 00/10090

(43) International Publication Date: 24 February 2000 (24.02.00)

(21) International Application Number: PCT/US99/18321

(22) International Filing Date: 12 August 1999 (12.08.99)

(30) Priority Data:
09/134,548 17 August 1998 (17.08.98) US

(71) Applicant: SUN MICROSYSTEMS, INC. [US/US]; 901 San Antonio Road, MS PAL01-521, Palo Alto, CA 94303 (US).

(72) Inventors: AGESEN, Ole; 15 Rolling Ridge Road, Franklin, MA 02038 (US). DETLEFS, David, L.; 94 Depot Street, Westford, MA 01886 (US). WHITE, Derek, R.; 54 Dana Road, Reading, MA 01867 (US).

(74) Agents: GARRETT, Arthur, S.; Finnegan, Henderson, Farabow, Garrett & Dunner, L.L.P., 1300 I Street, N.W., Washington, DC 20005-3315 (US) et al.

(81) Designated States: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published

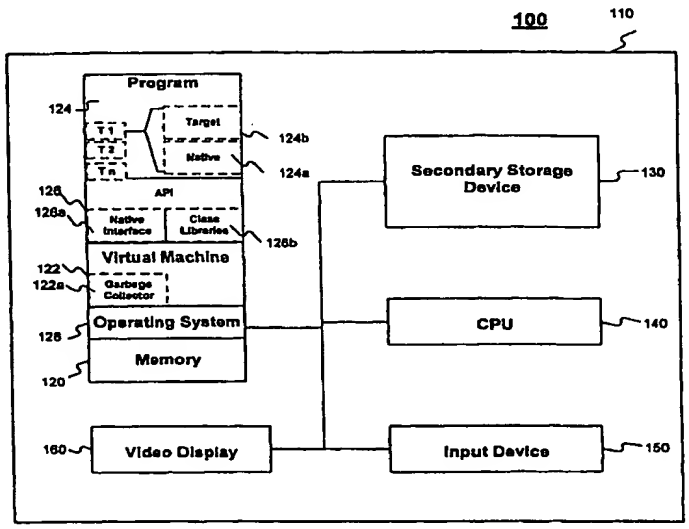
With international search report.

Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: METHOD, APPARATUS, AND ARTICLE OF MANUFACTURE FOR FACILITATING RESOURCE MANAGEMENT FOR APPLICATIONS HAVING TWO TYPES OF PROGRAM CODE

(57) Abstract

Methods, systems, and articles of manufacture consistent with the present invention provide a program component including a set of instructions native to the system, include in the set of native instructions an intruction to maintain information on use of a particular object, and permit reuse of memory resources corresponding to the particular object based on an indication from a source that the particular object is no longer being used, the source being different from any source used to provide information on use of objects associated with non-native instructions of the program component. Additionally, garbage collection is not permitted during native code operations to read or write data in object fields because during such operations an indication exists that such collection may be inaccurate and could possibly reclaim or relocate objects referenced by native code though not specified as such in the native code stack and global variables.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

METHOD, APPARATUS, AND ARTICLE OF MANUFACTURE FOR FACILITATING RESOURCE MANAGEMENT FOR APPLICATIONS HAVING TWO TYPES OF PROGRAM CODE

BACKGROUND OF THE INVENTION

A. Field of the Invention

This invention generally relates to memory management for computer systems and, more particularly, to a methodology for managing memory resources for an application program having two types of program code, native code executing directly in an operating environment and target code for execution by an abstract computing machine associated with the operating environment and responsible for memory management for both types of code.

B. Description of the Related Art

Object-oriented programming techniques have revolutionized the computer industry. For example, such techniques offer new methods for designing and implementing computer programs using an application programming interface (API) associated with a predefined set of "classes," each of which provides a template for the creation of "objects" sharing certain attributes determined by the class. These attributes typically include a set of data fields and a set of methods for manipulating the object.

The Java™ Development Kit (JDK) from Sun Microsystems, Inc., for example, enables developers to write object-oriented programs using an API with classes defined using the Java™ programming language. The Java programming language is described, for example, in a text entitled "The Java Language Specification" by James Gosling, Bill Joy, and Guy Steele, Addison-Wesley, 1996. The class library associated with the Java API defines a hierarchy of classes

with a child class (*i.e.*, subclass) inheriting attributes (*i.e.*, fields and methods) of its parent class. Instead of having to write all aspects of a program from scratch, programmers can simply include selected classes from the API in their programs and extend the functionality offered by such classes as required to suit the particular needs of a program. This effectively reduces the amount of effort generally required for software development.

The JDK also includes a compiler and a runtime environment with a virtual machine (VM) for executing programs. In general, software developers write programs in a programming language (in this case the Java programming language) that use classes from the API. Using the compiler, developers compile their programs into "class files" containing instructions for an abstract computing model embodied by the Java VM; these instruction are often called "bytecodes." The runtime environment has a class loader that integrates the class files of the application with selected API classes into an executable application. The Java VM then executes the application by simulating (or "interpreting") bytecodes on the host operating system/computer hardware. The Java VM thus acts like an abstract computing machine, receiving instructions from programs in the form of bytecodes and interpreting these bytecodes. (Another mode of execution is "just in time" compilation in which the VM dynamically compiles bytecodes into so-called native code for faster execution.) Details on the VM for the JDK can be found in a text entitled "The Java Virtual Machine Specification," by Tim Lindholm and Frank Yellin, Addison Wesley, 1996.

The Java VM also supports multi-threaded program execution. Multi-threading is the partitioning of a computer program or application into logically

independent "threads" of execution that can execute in parallel. Each thread includes a sequence of instructions to carry out a particular program task, such as a method for computing a value or for performing an input/output function. When employing a computer system with multiple processors, separate threads may execute concurrently on each processor.

Thus, object-oriented facilities like the JDK assist both development and execution of object-oriented systems. First, they enable developers to create programs in an object-oriented programming language using an API. Second, they enable developers to compile their programs, and third, they facilitate program execution by providing a virtual machine implementation.

However, object-oriented programs may not be suitable for all functions of a system or it may not be economically feasible to convert all of the programs in an existing legacy system into object-oriented programs. It may also be necessary, for a system having primarily object-oriented programs, to use features of a platform's operating system that are not available in implementations using a VM like the Java VM. Finally, the virtual machine implementation itself is generally not written in the language it executes but rather in the native code of the host machine. Thus, it is not uncommon for systems to have programs with "native" and "non-native" code.

For purposes of this description, native code includes code written in any programming language that is then compiled to run on a compatible operating system/hardware configuration. For example, native code in this context includes program code written in the C or C++ programming language and compiled by an appropriate compiler for execution on a particular platform, such as a computer

having the Windows 95 operating system running on an Intel Pentium processor. Native code is distinguishable from the non-native code, which will be referred to as "target code," because while non-native code is foreign to a platform's operating system/hardware configuration, its target for purposes of this description is an abstract computing machine, such as a VM, operating on any compatible platform configuration. For example, target code for the Java VM is generally written in the Java programming language. This combination of native and target code in the same application tends to complicate the management of memory resources (*i.e.*, the allocation and deallocation of memory) for such systems.

In practice, when an application seeks to refer to an object, the computer must first allocate or designate memory for the object. Using a "reference" to the allocated memory, the application can then properly manipulate the object. One way to implement a reference is by means of a "pointer" or "machine address," which uses multiple bits of information, however, other implementations are possible. Objects can themselves contain primitive data items, such as integers or floating point numbers, and/or references to other objects. In this manner, a chain of references can be created, each reference pointing to an object which, in turn, points to another object. When no chain of references in an application reaches a given object, the computer can deallocate or reclaim the corresponding memory for reuse.

Memory reclamation can be handled explicitly by the application program. This method, however, requires programmers to design programs to account for all allocated objects and to determine when the objects are available for reclamation. The alternative is to assign responsibility for memory management

to a runtime system responsible for controlling program execution. The Java VM, one such system responsible for controlling program execution for example, includes a "garbage collector" to manage available memory resources used during execution of Java code.

"Garbage collection" is the term used to refer to a class of algorithms used to carry out memory management, specifically, automatic reclamation. Garbage collection algorithms generally determine reachability of objects from the references held in some set of roots. When an object is no longer reachable, the memory that the object occupies can be reclaimed and reused. There are many known garbage collection algorithms, including reference counting, mark-sweep, and generational garbage collection algorithms. These, and other garbage collection techniques, are described in detail in a book entitled "Garbage Collection, Algorithms For Automatic Dynamic Memory Management" by Richard Jones and Raphael Lins, John Wiley & Sons, 1996.

To be effective, garbage collection techniques should be able to, first, identify references that are directly accessible to the executing program, and, second, given the reference to an object, identify references contained within that object, thereby allowing the garbage collector to transitively trace chains of references. Unfortunately, many of the described techniques for garbage collection have specific requirements which cause implementation problems, particularly when a garbage collector is charged with managing memory for a system having programs written in both native and target code. For example, the Java VM's garbage collector manages resources for Java code with relative ease;

however, it requires additional facilities to manage resources for other native code and even then the garbage collector has significant limitations.

In most language implementations, including the implementation of the Java programming language embodied in the JDK, stacks form one component of the root set. A stack is a region of memory in which stack frames may be allocated and deallocated. In typical object-oriented systems, each method executing in a thread of control allocates a stack frame, and uses the slots of that stack to hold the values of local variables. Some of those variables may contain references to heap-allocated objects. (The heap is an area of memory designated for resources associated with objects.) Such objects must be considered reachable as long as a method is executing. The term stack is used because the stack frames obey a last-in/first-out allocation discipline within a given thread of control. There is generally a stack associated with each thread of control, and when a thread involves both native and target program code, there are often two stacks, one for each type of code. Another component of the root set includes global variables used to hold references to objects outside a stack frame, which makes the objects available to multiple methods.

A garbage collector may be exact or conservative in how it treats different sources of references, such as stacks. A conservative collector knows only that some region of memory (*e.g.*, a slot for a local variable in the stack frame or a memory location holding a global variable) may contain references, but does not know whether or not a given value in that region is a reference. If such a collector encounters a value that is a possible reference value, it must keep the referenced object alive. Because of the uncertainty in recognizing references, the collector is

constrained not to move the object, since that would require updating the reference, which might actually be an unfortunately-valued integer or floating-point number. The main advantage of conservative collection is that it allows garbage collection to be used with systems not originally designed to support collection. For example, the collectors described in Bartlett, Joel F., Mostly-Copying Collection Picks Up Generations and C++, Technical Report TN-12, DEC Western Research Laboratory, October 1989, and Boehm, Hans Juergen and Weiser, Mark, Garbage Collection in an Uncooperative Environment. *Software-Practice & Experience,* 18(9), p. 807-820, September 1988, use conservative techniques to support collection for C and C++ programs.

In contrast, a collector is exact in its treatment of a memory region if it can accurately distinguish references from non-reference values in that region. Exactness has several advantages over conservatism. A conservative collector may retain garbage referenced by a non-reference value that an exact collector would reclaim. Perhaps more importantly, an exact collector is always free to relocate objects since it is able to identify references exactly. In an exact system, one in which references and non-references can be distinguished, this enables a wide range of useful and efficient garbage-collection techniques that cannot easily be used in a conservative setting. For example, the ability to relocate objects enables an exact collector to compact used memory during a collection cycle. However, a drawback of exact systems is that they must provide the information that makes them exact, *i.e.*, information on whether a given value in memory is a reference or a primitive value. A VM can do this effectively for its target code using techniques such as stack maps that distinguish references from primitive

values in the target code's stack. However, there is no known implementation that uses exact garbage collection for programs including both native and target code and allows the same level of flexibility and convenience in writing natuve code.

Sun Microsystems, Inc. also developed an interface, called the Java™ Native Interface (JNI), for native program code executing within the Java VM. The JNI is comprised of a library of functions, *i.e.*, an API, and developers of native code call upon these functions with references to them by name in the native code. The JNI functions enable the Java VM's garbage collector to obtain certain information concerning the native code for purposes of garbage collection. Using JNI functions, for example, the native code can reference objects in a heap managed by the Java VM's garbage collector. While the interface itself allows an implementation supporting exact garbage collection, in the most common implementation exact garbage collection is not possible. This is because references are maintained in the same stack used to hold references for the Java code and the Java VM uses an indicator in a special frame of Java code stack to control garbage collection of the native code objects. This implementation is satisfactory for conservative garbage collection but it does not prevent the "leaking" of direct object references outside the JNI stack frame. In other words, direct references to objects may be lost during a garbage collection cycle when all of the references may not be located in the JNI stack frame. Consequently, such an implementation of the JNI does not support an exact collection algorithm.

There is, therefore, a need for a mechanism that facilitates flexible garbage collection for memory resources for an application having two types of program code, native code familiar to an operating environment and target code for

execution by an abstract computing machine associated with the operating environment.

SUMMARY OF THE INVENTION

Methods, systems, and articles of manufacture consistent with the present invention, as embodied and broadly described herein, manage memory resources corresponding to objects in a system, by providing a program component including a set of instructions native to the system. These instructions include an instruction to maintain information on use of a particular object, and permit reuse of memory resources corresponding to the particular object based on an indication from a source that the particular object is no longer being used, the source being different from any source used to provide information on use of objects associated with non-native instructions of the program component. The system includes a runtime environment for executing the program component and a garbage collector of the runtime environment is invoked to permit reuse of memory resources. The garbage collector may implement an exact garbage collection algorithm. The source may be a stack or linked list associated with the native instructions.

In another implementation, memory resources corresponding to objects in a system are managed by providing a program component including a set of instructions native to the system. These instructions include an instruction to synchronize a garbage collector with the set of native instructions, and prevent, in response to the synchronize instruction, the garbage collector from permitting reuse of memory resources corresponding to a particular object until after operation of certain native instructions. The instruction to synchronize a garbage

collector with the set of native instructions may include setting an inconsistency bit in a data structure associated with the program component.

In yet another implementation, memory resources corresponding to objects in a system are managed by receiving a program component including instructions native to the system and instructions targeted to the abstract computing machine, wherein the instructions include references to objects representing memory resources, managing object references for the target instructions is a data structure, and managing object references for the native instructions in a linked list distinct from the data structure for managing object references for the target instructions. A garbage collection process is invoked to reclaim memory resources corresponding to objects based on information from both the data structure for object references for the target instructions and the linked list for object references for the native instructions. The garbage collection process reclaims the memory resources for a particular object when an indication exists that memory resources must be reclaimed to implement an instruction to allocate another object and it is determined that the target instructions and the native instructions no longer require the memory resources for the particular object.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate an implementation of the invention and, together with the description, serve to explain the advantages and principles of the invention. In the drawings,

FIG. 1 is a block diagram of an exemplary system with which methods, systems, and articles of manufacture consistent with the invention may be implemented;

FIG. 2 is a block diagram showing data structures for a multi-threaded application consistent with the present invention;

FIG. 3 is a block diagram showing a target stack and a native stack in accordance with the principles of the invention;

FIG. 4 is a block diagram showing a stack map for target code;

FIG. 5 is a block diagram showing a linked list of local roots created in accordance with the principles of the present invention;

FIG. 6 is a flowchart of the procedure for creating the linked list of FIG. 5 in a manner consistent with the principles of the present invention;

FIG. 7 is a flowchart illustrating a first method for synchronizing inconsistent threads with garbage collection in a manner consistent with the principles of the present invention; and

FIG. 8 is a flowchart showing the processing performed by a garbage collector in a manner consistent with the principles of the present invention.

DETAILED DESCRIPTION

Reference will now be made in detail to an implementation consistent with the present invention as illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings and the following description to refer to the same or like parts.

Introduction

Methods, systems, and articles of manufacture consistent with the present invention facilitate a flexible approach for garbage collection associated with the execution of systems having both native and target code by tracking objects referenced by each type of code in separate stacks. A stack obeys the Last-In-First-Out (LIFO) model and holds local variables, including references to objects in the heap. In contrast, "static" or global variables, which may also include references to objects in the heap, are managed outside the stack. The target code objects are identified using a map of object references in the stack for the target code, whereas objects referenced by native code are identified in the native stack using a linked list.

Additionally, garbage collection is not permitted during native code operations to read or write data in object fields because, during such operations, an indication exists that such collection may be inaccurate and could possibly fail to find and update object references in native code but not specified as such in the native code stack and global variables.

System Architecture

Figure 1 depicts an exemplary data processing system 100 suitable for practicing methods and implementing systems and articles of manufacture consistent with the present invention. Data processing system 100 includes a computer system 110 connected to a network 170, such as a Local Area Network, Wide Area Network, or the Internet.

Computer system 110 contains a main memory 120, a secondary storage device 130, a central processing unit (CPU) 140, an input device 150, and a video display 160, each of which are electronically coupled to the other parts. Main

memory 120 contains an operating system 128, a virtual machine (VM) 122, and a multi-threaded program 124. An exemplary VM 122 for purposes of this description is the Java VM described above. In such exemplary implementations, the Java VM is part of a runtime system, which also includes an API and other facilities required for running applications using the Java VM.

One skilled in the art will appreciate that although one implementation consistent with the present invention is described as being practiced using the Java VM, systems and methods consistent with the present invention may also be practiced in different environments, including those compatible with the Java VM. Also, although aspects of one implementation are depicted as being stored in memory 120, one skilled in the art will appreciate that all or part of systems and methods consistent with the present invention may be stored on or read from other computer-readable media, such as secondary storage devices, like hard disks, floppy disks, and CD-ROM; a carrier wave received from a network such as the Internet; or other forms of ROM or RAM. Finally, although specific components of data processing system 100 have been described, one skilled in the art will appreciate that a data processing system suitable for use with the exemplary embodiment may contain additional or different components.

VM 122 includes a garbage collector 122a. In one implementation, garbage collector 122a implements an exact garbage collection algorithm, although other algorithms may be implemented without departing from the principles consistent with the present invention.

For purposes of simplifying the illustration program 124 is shown with more than one thread of execution T_1, T_2 and T_n, although those skilled in the art

will understand that each thread represents a process consisting of a set of program instructions executing in CPU 140. A single thread such as T_1, may execute portions of native code 124a and target code 124b. Consequently, VM 122 and garbage collector 122a must manage memory resources for both types of code operating in the same thread.

API 126 includes native interface 126a and class libraries 126b. Class libraries 126b includes a set of classes, and developers can select classes for target code 124b. Native interface 126a includes a set of functions, and developers can include in native code 124a calls to the native interface 126a to effect various functions, including memory resource management in a manner consistent with principles of the present invention.

In general, native interface 126a includes instructions to perform two types of functions. The first concerns managing a native stack associated with a thread of control including certain portions of native code 124a. These functions build and maintain a linked list structure within the native stack to identify all stack entries containing references to objects. Garbage collector 122a traverses the list to identify all such referenced objects and, if an object in the heap is not referenced in the native stack or elsewhere, for example, in a global variable or a stack for the target code, then garbage collector 122a reclaims the object's resources. The second set of functions in native interface 126a enable native code developers to specify periods of time during execution of native code 124a for which garbage collection is not permitted because, for example, execution of a garbage collection cycle may destroy objects for which references may exist outside of the known reference sources, including the native stack, global

variables, and target code stack, or move such objects, updating only the known but not the unknown references. If garbage collection were permitted during such periods, there is a risk of loss of objects having valid references and, potentially, later program execution errors.

Appendix A contains a document entitled the "LLNI User's Guide," which details how to use an exemplary interface consistent with the principles of the present invention with an implementation of the Java VM having a collector that uses an exact garbage collection algorithm.

Stack Structures

VM 122 is responsible to executing program 124 using CPU 140 in conjunction with an operating system 128. In alternative configurations, all or some of the functions of VM 122 may be incorporated in the operating system or CPU 140. To facilitate program execution in a multi-threaded fashion, VM 122 maintains a thread element for each thread. As shown in Fig. 2, a thread element 246 includes two stacks 250 and 252 for managing execution of each type of code. For example, target stack 250 holds references to objects in memory used by target code 124b and native stack 252 holds references to objects in memory used by native code 124a. Those skilled in the art will recognize that implementations consistent with the principles of the present invention may involve intermingling the two stacks in a memory structure or use a single stack with appropriate designations to distinguish between portions of the stack used for target code versus those portions used for native code.

Thread data elements are typically linked together by thread data structures, such as thread data structure 248. For example, thread 1 data structure

248 identifies or points to a location for the data structure for a next thread. By linking all thread data elements together, VM 122 may step from thread to thread, and access the data elements of each thread.

VM 122 also maintains global roots 260 and native global roots 262 separate from the thread elements. Global roots 260 and native global roots 262 contain variables accessible by all threads. These variables may include references to stored objects.

Figure 3 is a block diagram showing a portion of thread data element 246 of Fig. 2 in greater detail. Thread data element 246 comprises thread data structure 248, target stack 250 and native stack 252. Thread data structure 248 includes fields of information for managing the data elements of the thread. For example, thread data structure 248 stores inconsistency bit 310, target code stack pointer 312, native code stack pointer 314, and next thread pointer 315. Thread data structure 248 may also store other information about the thread. Inconsistency bit 310 is set whenever a thread is starting to enter a region of program code that may result in pointers being used in a manner inconsistent with the implemented garbage collection algorithm.

Target code stack pointer 312 points to target stack 250. Target stack 250 stores information used during execution of target code of a particular thread. Similarly, native code stack pointer 314 points to native stack 252, which stores information used during execution of native code of a particular thread. Target stack 250 and native stack 252 have similar structures. Target stack 250 comprises frames 316, 318 and 320. Native stack 252 comprises frames 322 and

324. Finally, next thread pointer 315 points to the thread data structure of the next thread, as explained with reference to Fig. 2.

In one implementation, each stack frame 316, 318 and 320 for target code includes an area for holding local variables, an invoking frame pointer, a method pointer, a program counter (PC), and an operand stack, as shown in Fig. 3. When a target or native method starts executing, a stack frame for the method is added on the appropriate stack. The stack frame holds variables used during execution of the method, including references to stored objects.

A target method uses values in either the operand stack or the local variables of the stack frame from which it is executing. For example, a target method might add two integers by pushing the integers on the operand stack and then performing an add bytecode which pops the top two items off the operand stack, adds them, and pushes the answer back on.

Each target code stack frame 320, 318, and 320 also contains an invoking frame that points to the previous stack frame on the stack, a method pointer that points to a method block associated with the method, and a program counter (PC) that points to the current line of the method being executed.

In contrast, native code stack 252 comprises a series of linked frames 322 and 324, each of which holds local variables used by the native method executing out of the frame. As in target stack 250, native stack 252 contains local variables that reference stored objects.

As target or native methods execute, VM 122 manipulates local variables referencing stored objects, deleting references to stored objects when no longer needed by the method. Even though an object is no longer needed, it still

occupies space in the heap. As more object references are deleted, the space occupied by unused objects grows until there is no space left in the heap for allocating new objects. To provide more space for object allocation, garbage collector 122a periodically determines which objects are being referenced, and reclaims the remaining space in the heap. The reclaimed space can then be used for allocating space for more objects. Garbage collector 122a determines which objects are being referenced by stepping through each thread data element 246 using the thread data structure 248 of each element.

An object is usually considered referenced if there is some path of pointers leading to the object from roots located in variables of the program. The roots of a program at a given point in execution are comprised of the global (i.e. static) variables of the program together with the local variables of any procedure or method currently being executed at that execution point. This includes, for example, global roots 260 and native global roots 262 of Fig. 2, and the local variables of stack frames 316, 318 and 320 of Fig. 3.

It is generally not difficult to identify the static variables of a program, and trace objects from those containing pointers because static variables implementing pointers generally remain pointers throughout execution. It can, however, be difficult to identify which local variables contain pointers to objects as opposed to primitive values.

To save stack space, for example, the slots in stack frames are sometimes reused. Consider a method "m" that has two subparts: a pointer-containing variable "p" that is used only during the first part, and an integer-containing variable "i" that is used only during the second part. Since "p" and "i" are never in

use at the same time, a single slot "s" in a stack frame for "m" might be used for both. In such a situation, garbage collector 122a has difficulty determining whether to consider slot "s" a pointer or a primitive. If it does not consider "s" a pointer, and "s" actually does contain a pointer, the garbage collector risks incorrectly recycling the object to which "s" points.

Garbage Collection for Target Code

To address this problem for target code, thread element 246 has a corresponding stack map, as shown in Fig. 4. To create a stack map for a method, the method is first scanned to find safe points for garbage collection. Typically, these garbage collection safe points are times of transition, such as at a call or backward branch instruction. Once a safe point is found, the stack map defining all of the pointer locations is generated and associated with that particular instruction. Therefore, when a safe point is reached during execution, a garbage collector can determine from the stack map where each pointer is located in the stack frame at the time the respective instruction is executed. Using this information, the garbage collector knows exactly where all pointers are located. Stack maps can be generated at any point before garbage collection. For example, they can be generated when the program is compiled or during program execution.

Figure 4 is a block diagram illustrating an example of a stack map. In the stack frame 410 associated with a method of thread n, method pointer 412 points to method block 414. Method block 414 points to the method 416, which is the code of the method. Method block 414 also points to stack map data structure 418. Stack map data structure 418 comprises program counter values corresponding to particular lines of the code in method 416. Each program

counter value is associated with a respective map in the set 420. Each map in the set of stack maps 420 specifies the stack slots or memory registers containing references to heap-allocated objects. For purposes of illustration, each stack map is divided into two sections, indicated by a heavy vertical line. Locations to the left of the heavy line indicate slots (S1, S2, S3, S4) in the stack frame of the method, and locations to the right indicate registers (R1, R2).

As described above with reference to Fig. 3, a PC is stored in each frame, and is used to track the line currently being executed in the method corresponding to the stack frame. When the PC in stack 410 (not shown) equals line 10, stack map 422 defines which slots and registers have object pointers at that particular point in execution of method 416. Stack map 422 indicates that slots S1 and S4, and register R1, each marked by a "1," have a pointer when execution of method 416 is at program counter value 10. Slots S2 and S3, and register R2, each marked by an "O," do not contain pointers. Therefore, the garbage collector can determine precisely where each pointer is located for method 416 by using the set of stack maps 420.

When garbage collector 122a begins a garbage collection cycle, each thread is stopped and advanced to a safe point. Once execution reaches the safe point, garbage collector 122a uses the stack map associated with each method to determine pointer locations with certainty.

To find the stack map associated with a particular method, garbage collector 122a first steps through each thread data structure to access the target stacks, and uses the method pointer in the stack frame to access the corresponding set of stack maps. Garbage collector 122a then uses the stack map corresponding

to the line of code at which the method was stopped to determine the stack frame locations having pointers referencing objects. Further details on the use of a stack map in this fashion for garbage collection can be found in O. Agesen, D. Detlefs, J.E.B. Moss, "Garbage Collection and Local Variable Type-Precision and Liveness in Java™ Virtual Machines," Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, ACM, 1998, pp. 269-279, which is incorporated herein by reference.

Garbage Collection for Native Code

In contrast, garbage collection for code operating out of a native stack in a manner consistent with the principles of the present invention involves including in the native code calls to certain functions of native interface 126a. Developers can modify existing native code to include the function calls; alternatively, they can write new native code with the function calls. For purposes of this description, there are two types of function calls. The first concerns creating and maintaining a linked list in the stack for the native code identifying all of the slots for local variables containing references to objects in the heap. The second involves setting inconsistency bit 310. When set, this bit prevents garbage collector 122a from executing during an "unsafe" period when not all local variables containing references to objects are identifiable.

Linked List

Figure 5 is a block diagram showing an implementation of a per-thread linked list of local roots created in accordance with the principles of the present invention. Stack 514 contains stack frames having slots used by native methods during execution of the methods. Slots B, D and F point to objects. Slot E points

to slot D, and slot C points to slot B. Native stack pointer 512 points to stack 514, and local variables pointer (LVP) 510 points to the first local variable in the linked list.

The linked list is formed by grouping each object pointer with a pointer to the previous object pointer in the stack. Therefore, by using the value of LVP 510, garbage collector 122a can determine exactly the location of the object pointer in slot F, go to the next location and trace slot E to the object pointer in slot D, and go to the next location and trace slot C to the object pointer in slot B. Slot B is followed by A, which has a null value, indicating the end of the linked list. In this way, the garbage collector can determine the exact location of each pointer in the stack.

Figure 6 is a flowchart showing how the linked list of Fig. 5 is created. For purposes of this description assume LVP 510 is pointing to slot D. Upon receiving a routine call from native code 124a that requires creation of a new object pointer, VM 122 uses code in native interface 126a to set the value of slot E to the value of LVP 510 (step 610). Thus, slot E now points to slot D. VM 122 then initializes the slot for the pointer to the new object in the heap (i.e., slot F) by setting the slot to a null value (step 612). VM 122 then sets the value of LVP 510 to the address of slot F (step 614), so that LVP 510 continues to point to the uppermost object pointer in the stack. Finally, VM 122 sets the value of slot F to the address of the object in heap 524 (step 616).

When the native routine is done with a local root, VM 122 pops it off the top of the local roots linked list by setting LVP 510 to the address of the next

object pointer downward in the stack. In the example shown, LVP 510 will be set to the value stored in slot E, i.e., the address of slot D.

Any global variables containing object references are also manipulated using a similar linked list structure.

GC Synchronization

Objects in the heap can be accessed by means of direct and indirect pointers. A direct pointer is the same as a reference in a local or global variable and can be used in a program to access an object. In contrast, an indirect pointer is a pointer to a direct pointer. Consistent regions of program code use only indirect pointers to reference objects by means of direct pointers. To access an object, for example, to write a value in a field of the object, the indirect pointer is "dereferenced," obtaining access to the direct pointer and thus access to the object itself. Regions of program code during which objects are accessed by using direct pointers are "inconsistent" regions because the dereferencing of an indirect pointer may copy a direct pointer value into a location not known by the garbage collector to contain such a pointer. Thus, garbage collection is not permitted during inconsistent regions of program code because it is not possible to determine exactly which slots in the stack frame are pointers to objects in the heap. If the garbage collector relocates an object (as is often the case with a compacting garbage collector, for example), the collector may fail to update direct pointers that were obtained by dereferencing indirect pointers to the new location of the relocated objects. Thus, to access an object for read or write purposes in a manner consistent with the present invention, garbage collection must be postponed until the access operation is completed. This is accomplished by including in the native

code a call to a GC synch routine of interface 126a to set inconsistent bit 310 in thread data structure 248, indicating that the thread is now in an inconsistent region and garbage collection is not permitted. For efficiency, the call may be "in-line," meaning the program code for the called routine is actually included in the calling program instead of requiring the routine to be loaded from another location. The GC synch routine synchronizes native code with garbage collector 122a.

Figure 7 is a flowchart illustrating a method for synchronizing inconsistent threads with garbage collection consistent with the principles of the present invention. When an inconsistent region of code is entered, the native code declares the thread inconsistent by calling a routine of interface 126a to set the inconsistent bit 310 (step 710).

VM 122 then dereferences the indirect pointer received in the routine call using interface 126a (step 712) to obtain the value pointed to by the indirect pointer (i.e., the direct pointer). After native code is through using the direct pointer associated with an indirect pointer, the thread consistency bit is reset (step 718). A global flag is then checked, to determine whether a garbage collection was requested while in the inconsistent region (step 720). If the global flag indicates that no garbage collection was requested, the process is exited. If the global flag indicates that garbage collection has been requested, VM 122 waits until collection is complete (step 722).

Figure 8 is a flowchart showing the processing performed by garbage collector 122. For example, when a method requests to allocate an object and the request cannot be granted because the heap is full, the thread must run a garbage

collection to find memory space to satisfy the allocation request. In response to a garbage collection request, garbage collector 122 first stops all threads currently being executed (step 810). Garbage collector 122a then determines whether all threads are consistent by checking the inconsistent bit 310 of each thread data structure 248 (step 812). If all threads are consistent, garbage collection is performed (step 820), after which the threads are restarted.

If some threads are not consistent, garbage collector 122 raises a global flag (step 814) indicating to threads coming out of inconsistent regions that the thread should synchronize with garbage collector 122 when it comes out of an inconsistent state. Garbage collector 122 then restarts the inconsistent threads (step 816) and waits until all threads are consistent (step 818). Upon all threads becoming consistent, garbage collection is performed (step 820) and the threads are restarted.

Therefore, threads may temporarily enter an inconsistent region, which prevents garbage collector 122 from starting a collection cycle.

Conclusion

Methods, systems, and articles of manufacture consistent with the present invention therefore facilitate a flexible approach for garbage collection associated with the execution of systems having both native and target code and permit implementations using either a conservative or exact collection algorithm.

The foregoing description of an implementation of the invention has been presented for purposes of illustration and description. It is not exhaustive and does not limit the invention to the precise form disclosed. Modifications and variations are possible in light of the above teachings or may be acquired from

practicing of the invention. For example, the described implementation includes software but the present invention may be implemented as a combination of hardware and software or in hardware alone. The invention may be implemented with both object-oriented and non-object-oriented programming systems. The scope of the invention is defined by the claims and their equivalents.

WHAT IS CLAIMED IS:

1. A method for managing memory resources corresponding to objects in a system, comprising:

executing a program component including a set of instructions native to the system, the set including an instruction to maintain information on use of a particular object; and

permitting reuse of memory resources corresponding to the particular object based on an indication from a source that the particular object is no longer being used, the source being different from any source used to provide information on use of objects associated with non-native instructions of the program component.

2. The method of claim 1, wherein the system includes a runtime environment for executing the program component, and wherein permitting reuse of memory resources includes

invoking a garbage collector of the runtime environment.

3. The method of claim 2, wherein invoking a garbage collector includes implementing an exact garbage collection algorithm.

4. The method of claim 1, wherein permitting reuse of memory resources includes

using a stack associated with the native instructions.

5. The method of claim 1, wherein permitting reuse of memory resources includes

using a linked list associated with the native instructions.

6. A method for managing memory resources corresponding to objects in a system, comprising:

executing a program component including a set of instructions native to the system, wherein the set of native instructions includes an instruction to synchronize a garbage collector with the program component; and

preventing, in response to execution of the synchronize instruction, the garbage collector from permitting reuse of memory resources corresponding to a particular object until after operation of certain native instructions.

7. The method of claim 6, wherein the instruction to synchronize a garbage collector with the set of native instructions includes

setting an inconsistency bit in a data structure associated with the program component.

8. The method of claim 6, further comprising:

signaling the garbage collector to permit reuse of memory resources corresponding to a particular object after operation of certain native instructions.

9. The method of claim 8, wherein the instruction to synchronize a garbage collector with the set of native instructions includes

setting an inconsistency bit in a data structure associated with the program component.

10. The method of claim 8, wherein signaling the garbage collector includes resetting the inconsistency bit.

11. A method for managing memory resources corresponding to objects in a system, comprising:

providing a program component including a set of instructions native to the system, wherein the set of native instructions includes an instruction to synchronize a garbage collector with the program component; and

signaling the garbage collector to postpone a start of a collection process until after operation of certain native instructions in response to the synchronize instruction.

12. A memory for managing memory resources corresponding to objects in a system, the memory comprising:

a set of instructions native to the system, the set including an instruction to maintain information on use of a particular object; and

an interface with code for permitting reuse of memory resources corresponding to the particular object based on an indication from a source that the particular object is no longer being used, the source being different from any source used to provide information on use of objects associated with non-native instructions.

13. A computer-implemented method for managing resources corresponding to objects in a memory, comprising:

executing a program component including a set of instructions native to the system, the set including an instruction to maintain information on use of a particular object; and

relocating another object in the memory based on an indication from a source that the particular object is no longer being used, the source being different from any source used to provide information on use of objects associated with non-native instructions of the program component.

14. The method of claim 13, further comprising:

updating a data structure referencing the relocated object with its new location.

15. A method for managing memory resources corresponding to objects in a system having an abstract computing machine, comprising:

executing a program component including first instructions native to the system and second instructions targeted to the abstract computing machine, wherein the first and second instructions include references to objects representing memory resources;

managing object references for the second instructions in a data structure; and

managing object references for the first instructions in a linked list distinct from the data structure for managing object references for the second instructions.

16. The method of claim 15, further comprising:

invoking a garbage collection process to reclaim memory resources corresponding to objects based on information from both the data structure for object references for the second instructions and the linked list for object references for the first instructions.

17. The method of 16, wherein invoking the garbage collection process includes

reclaiming the memory resources for a particular object when an indication exists that memory resources must be reclaimed to perform an instruction to allocate another object and it is determined that the second instructions and the first instructions no longer require the memory resources for the particular object.

18. A system for managing memory resources corresponding to objects, comprising:

a memory having a program component including a set of instructions native to the system, the set including an instruction to maintain information on use of a particular object; and

a processor configured to permit reuse of memory resources corresponding to the particular object based on an indication from a source that the particular object is no longer being used, the source being different from any source used to provide information on use of objects associated with non-native instructions of the program component.

19. The system of claim 18, wherein the processor is further configured to invoke a garbage collector.

20. The system of claim 19, wherein the garbage collector implements an exact garbage collection algorithm.

21. The system of claim 18, wherein the source is a stack associated with the native instructions.

22. The system of claim 18, wherein the source is a linked list associated with the native instructions.

23. A system for managing memory resources corresponding to objects, comprising:

a memory having a program component including a set of instructions native to the system, wherein the set of native instructions includes an instruction to synchronize a garbage collector with the set of native instructions; and

a processor configured to prevent, in response to the synchronize instruction, the garbage collector from permitting reuse of memory resources corresponding to a particular object until after operation of certain native instructions.

24. The system of claim 23, wherein the processor is further configured to set an inconsistency bit in a data structure associated with the program component.

25. The system of claim 23, wherein the processor is further configured to signal the garbage collector to permit reuse of memory resources corresponding to a particular object after operation of certain native instructions.

26. The system of claim 25, wherein the processor is further configured to set an inconsistency bit in a data structure associated with the program component.

27. The system of claim 25, wherein processor is further configured to reset the inconsistency bit.

28. A system for managing memory resources corresponding to objects, comprising:

a memory having a program component including a set of instructions native to the system, the set of native instructions including an instruction to synchronize a garbage collector with the set of native instructions; and

a processor configured to signal the garbage collector to postpone a start of a collection process until after operation of certain native instructions in response to the synchronize instruction.

29. A system for managing memory resources corresponding to objects, comprising:

a memory having a program component including a set of instructions native to the system, the set including an instruction to maintain information on use of a particular object; and

a processor configured to relocate another object based on an indication from a source that the particular object is no longer being used, the source being different from any source used to provide information on use of objects associated with non-native instructions of the program component.

30. The system of claim 29, wherein the processor is further configured to update a data structure referencing the relocated object with its new location.

31. The system of claim 29, wherein the processor is further configured to invoke a garbage collector.

32. A system for managing memory resources corresponding to objects and having an abstract computing machine, comprising:

a memory having a program component including first instructions native to the system and second instructions targeted to the abstract computing machine, wherein the first and second instructions use references to objects representing memory resources; and

a processor configured to manage object references for the second instructions in a data structure and object references for the first instructions in a linked list distinct from the data structure for managing object references for the target instructions.

33. The system of claim 32, wherein the processor is further configured to invoke a garbage collection process to reclaim memory resources corresponding to objects based on information from both the data structure for object references for the second instructions and the linked list for object references for the first instructions.

34. The system of 33, wherein the processor is further configured to reclaim the memory resources for a particular object when an indication exists that memory resources must be reclaimed to implement an instruction to allocate another object and it is determined that the second instructions and the first instructions no longer require the memory resources for the particular object.

35. A computer-readable medium containing instructions to perform a method for controlling a data processing system to manage memory resources corresponding to objects in the data processing system, the method comprising:

executing a program component including a set of instructions native to the system, the set including an instruction to maintain information on use of a particular object; and

permitting reuse of memory resources corresponding to the particular object based on an indication from a source that the particular object is no longer being used, the source being different from any source used to provide information on use of objects associated with non-native instructions of the program component.

36. The computer-readable medium of claim 35, wherein the data processing system includes a runtime environment for executing the program component, and wherein permitting reuse of memory resources includes

invoking a garbage collector of the runtime environment.

37. The computer-readable medium of claim 36, wherein invoking a garbage collector includes

implementing an exact garbage collection algorithm.

38. The computer-readable medium of claim 35, wherein the source is a stack associated with the native instructions.

39. The computer-readable medium of claim 35, wherein the source is a linked list associated with the native instructions.

40. A computer-readable medium containing instructions to perform a method for controlling a data processing system to manage memory resources corresponding to objects in the data processing system, the method comprising:

providing a program component including a set of instructions native to the system;

including in the set of native instructions an instruction to synchronize a garbage collector with the set of native instructions; and

preventing, in response to the synchronize instruction, the garbage collector from permitting reuse of memory resources corresponding to a particular object until after operation of certain native instructions.

41. The computer-readable medium of claim 40, wherein the instruction to synchronize a garbage collector with the set of native instructions includes

setting an inconsistency bit in a data structure associated with the program component.

42. The computer-readable medium of claim 40, wherein the method further comprises:

signaling the garbage collector to permit reuse of memory resources corresponding to a particular object after operation of certain native instructions.

43. The computer-readable medium of claim 42, wherein the instruction to synchronize a garbage collector with the set of native instructions includes

setting an inconsistency bit in a data structure associated with the program component.

44. The computer-readable medium of claim 42, wherein signaling the garbage collector includes

resetting the inconsistency bit.

45. A computer-readable medium containing instructions to perform a method for controlling a data processing system to manage memory resources corresponding to objects in the data processing system, the method comprising:

providing a program component including a set of instructions native to the system;

including in the set of native instructions an instruction to synchronize a garbage collector with the set of native instructions; and

signaling the garbage collector to postpone a start of a collection process until after operation of certain native instructions in response to the synchronize instruction.

46. A computer-readable medium containing instructions to perform a method for controlling a data processing system to manage memory resources corresponding to objects in the data processing system, the method comprising:

executing a program component including a set of instructions native to the system, the set including an instruction to maintain information on use of a particular object; and

relocating another object based on an indication from a source that the particular object is no longer being used, the source being different from any source used to provide information on use of objects associated with non-native instructions of the program component.

47. The computer-readable medium of claim 46, wherein the method further comprises:

updating a data structure referencing the relocated object with its new location.

48. A computer-readable medium containing instructions to perform a method for controlling a data processing system to manage memory resources corresponding to objects in the data processing system, the method comprising:

executing a program component including instructions native to the system and instructions targeted to the abstract computing machine, wherein the instructions include references to objects representing memory resources;

managing object references for the target instructions in a data structure; and

managing object references for the native instructions in a linked list distinct from the data structure for managing object references for the target instructions.

49. The computer-readable medium of claim 48, wherein the method further comprises:

invoking a garbage collection process to reclaim memory resources corresponding to objects based on information from both the data structure for object references for the target instructions and the linked list for object references for the native instructions.

50. The computer-readable medium of 49, wherein invoking the garbage collection process includes

reclaiming the memory resources for a particular object when an indication exists that memory resources must be reclaimed to implement an instruction to allocate another object and it is determined that the target instructions and the native instructions no longer require the memory resources for the particular object.

51. A system for managing memory resources corresponding to objects, comprising:

means for executing a program component including a set of instructions native to the system, the set including an instruction to maintain information on use of a particular object; and

means for permitting reuse of memory resources corresponding to the particular object based on an indication from a source that the particular object is no longer being used, the source being different from any source used to provide information on use of objects associated with non-native instructions of the program component.

52. A memory device encoded with a data structure used for managing system resources for a thread executing in a computer and including a set of instructions native to the computer and a set of instructions targeted to an abstract computer machine operating in the computer, the data structure comprising:

a first field with an identifier for a memory area for holding references to objects used in the native instructions;

a second field with an identifier for a different memory area for holding references to objects used in the target instructions; and

an inconsistency bit that, when set, prevents a garbage collector from permitting reuse of memory resources corresponding to a particular object until after operation of certain native instructions.

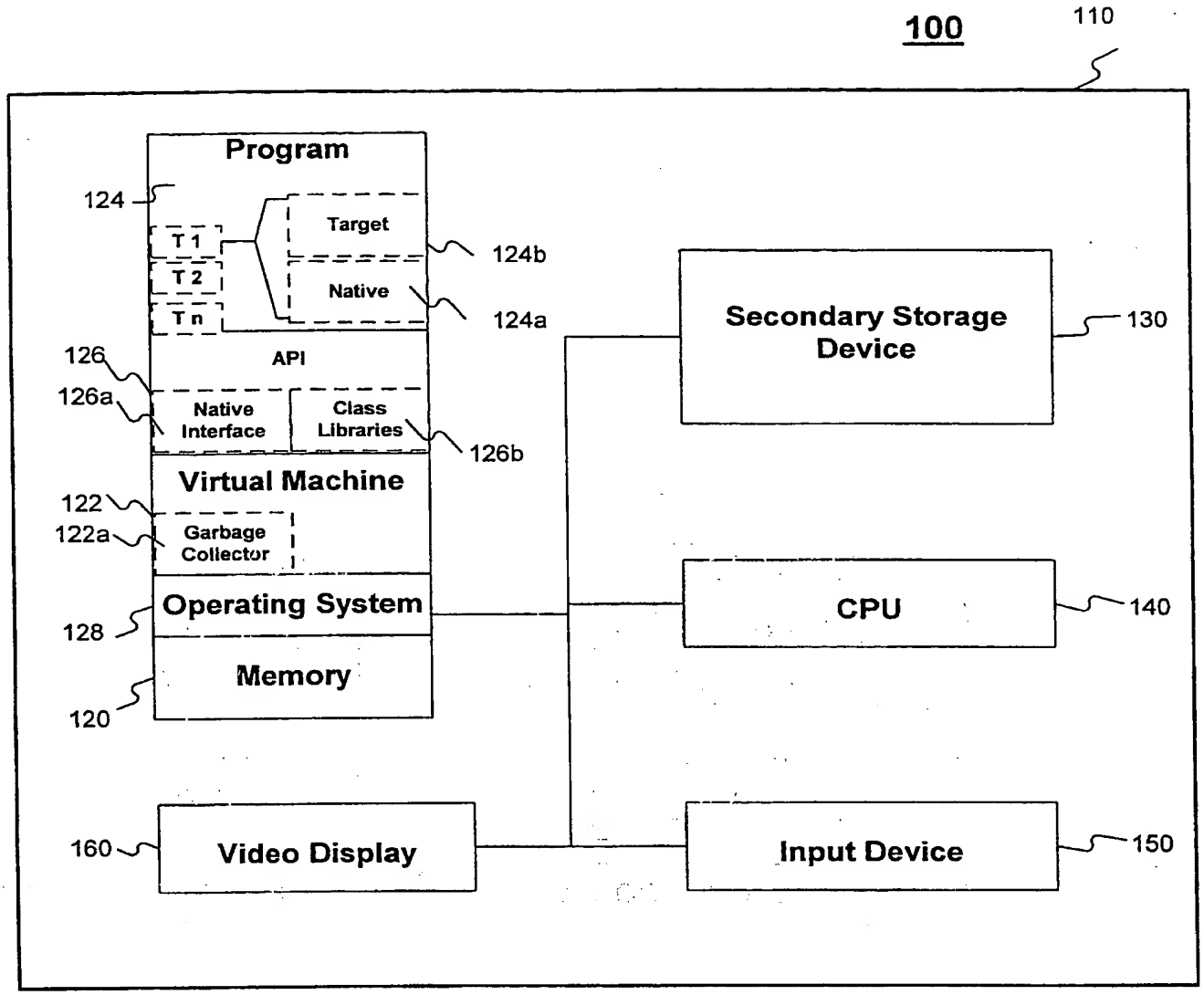


FIG. 1

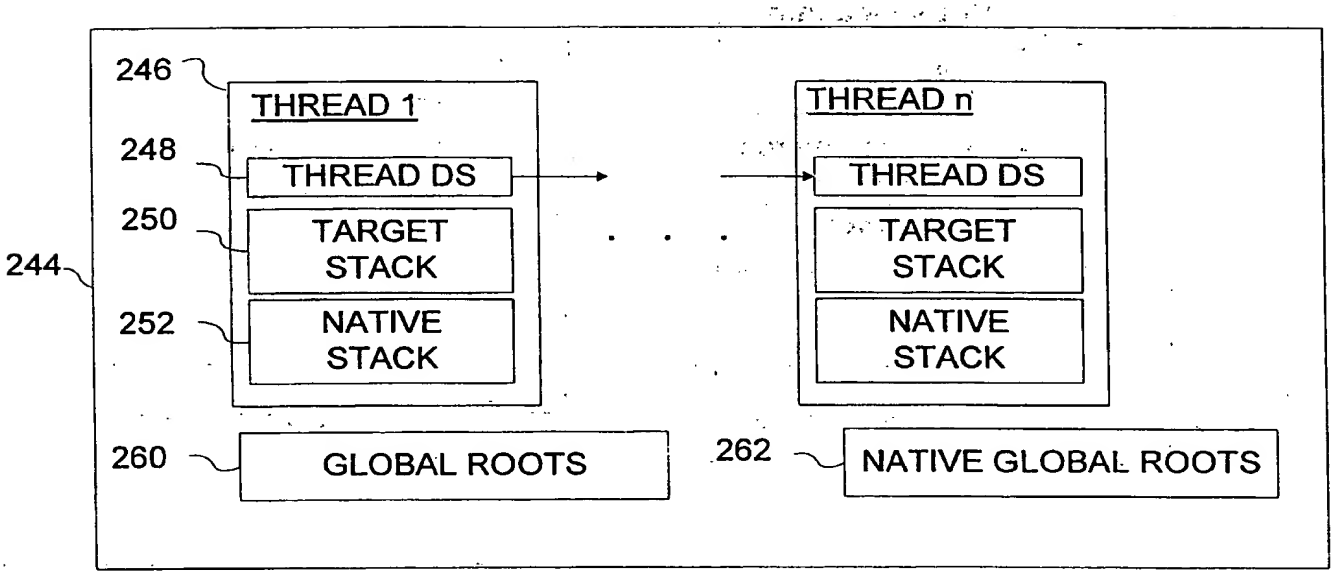


FIG. 2

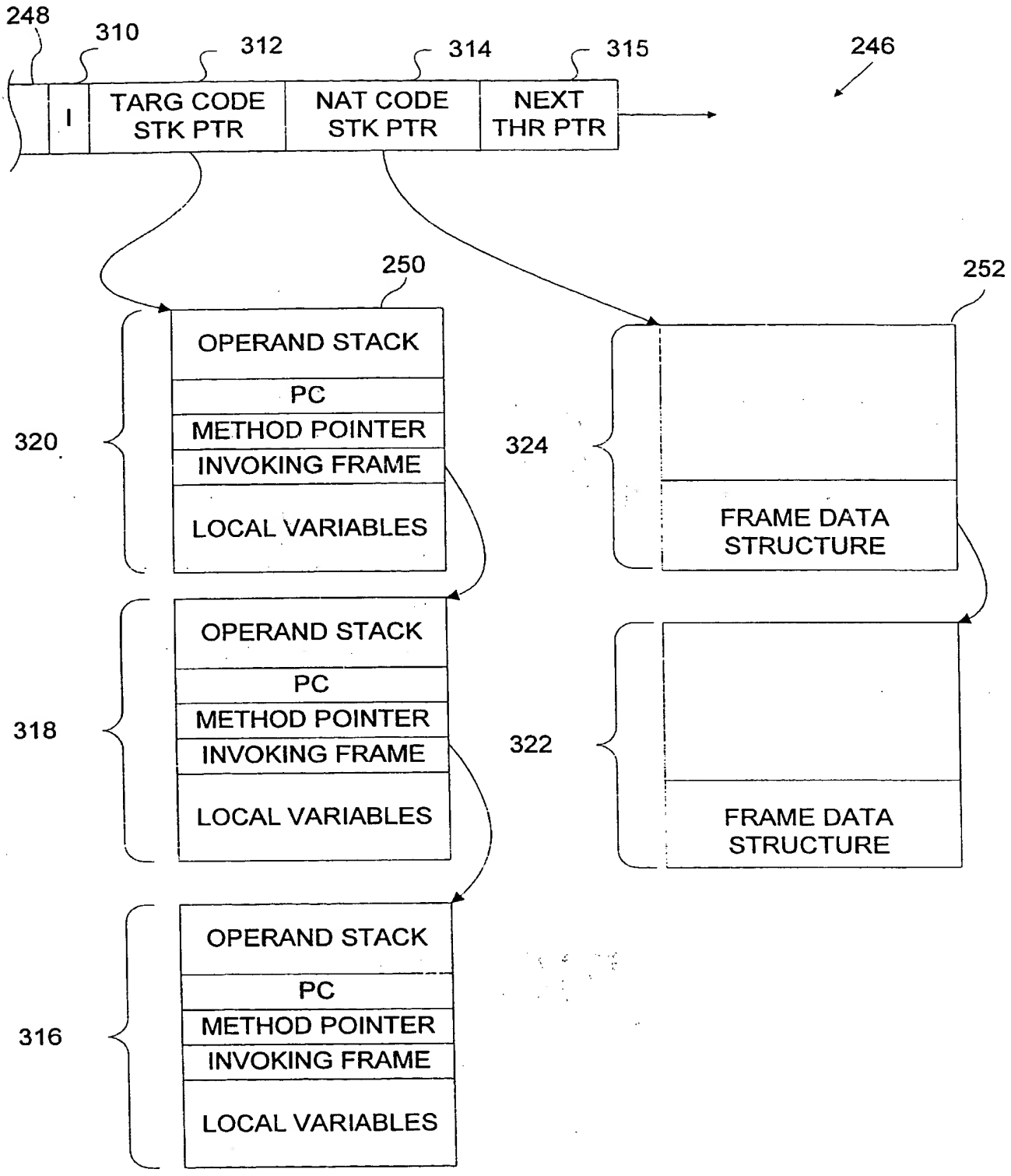


FIG. 3

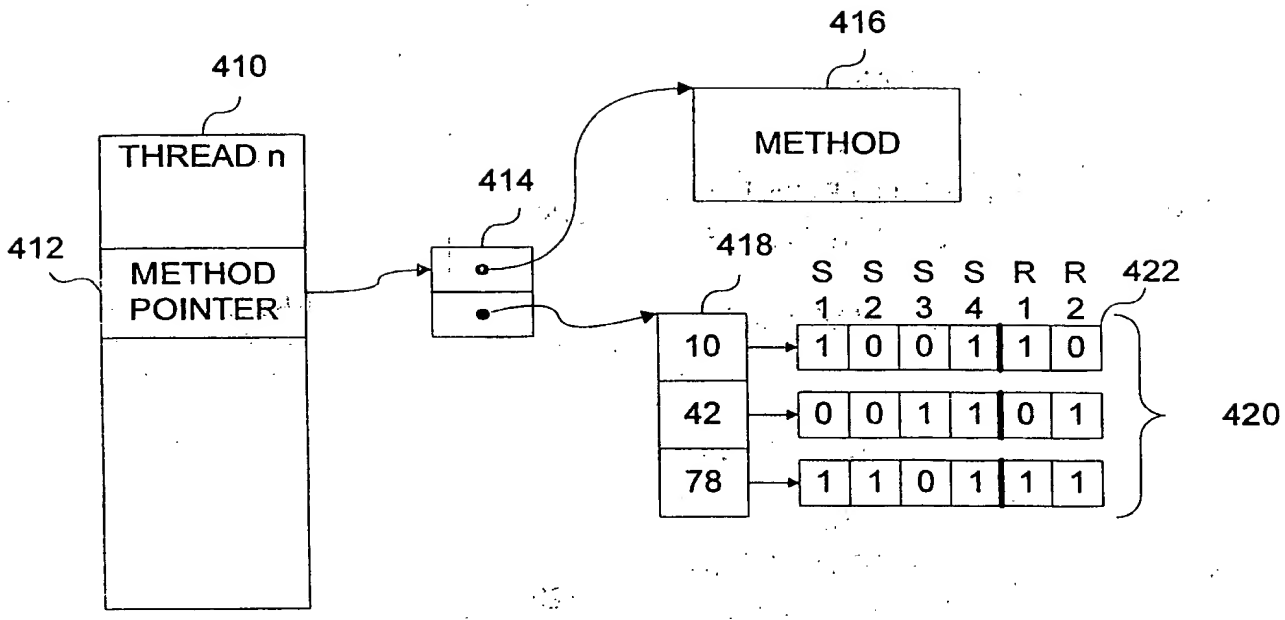

410
THREAD n
412
METHOD POINTER
414
416
METHOD
418
S1 S2 S3 S4 R1 R2
422
10
1 0 0 1 1 0
42
0 0 1 1 0 1
78
1 1 0 1 1 1
420

FIG. 4

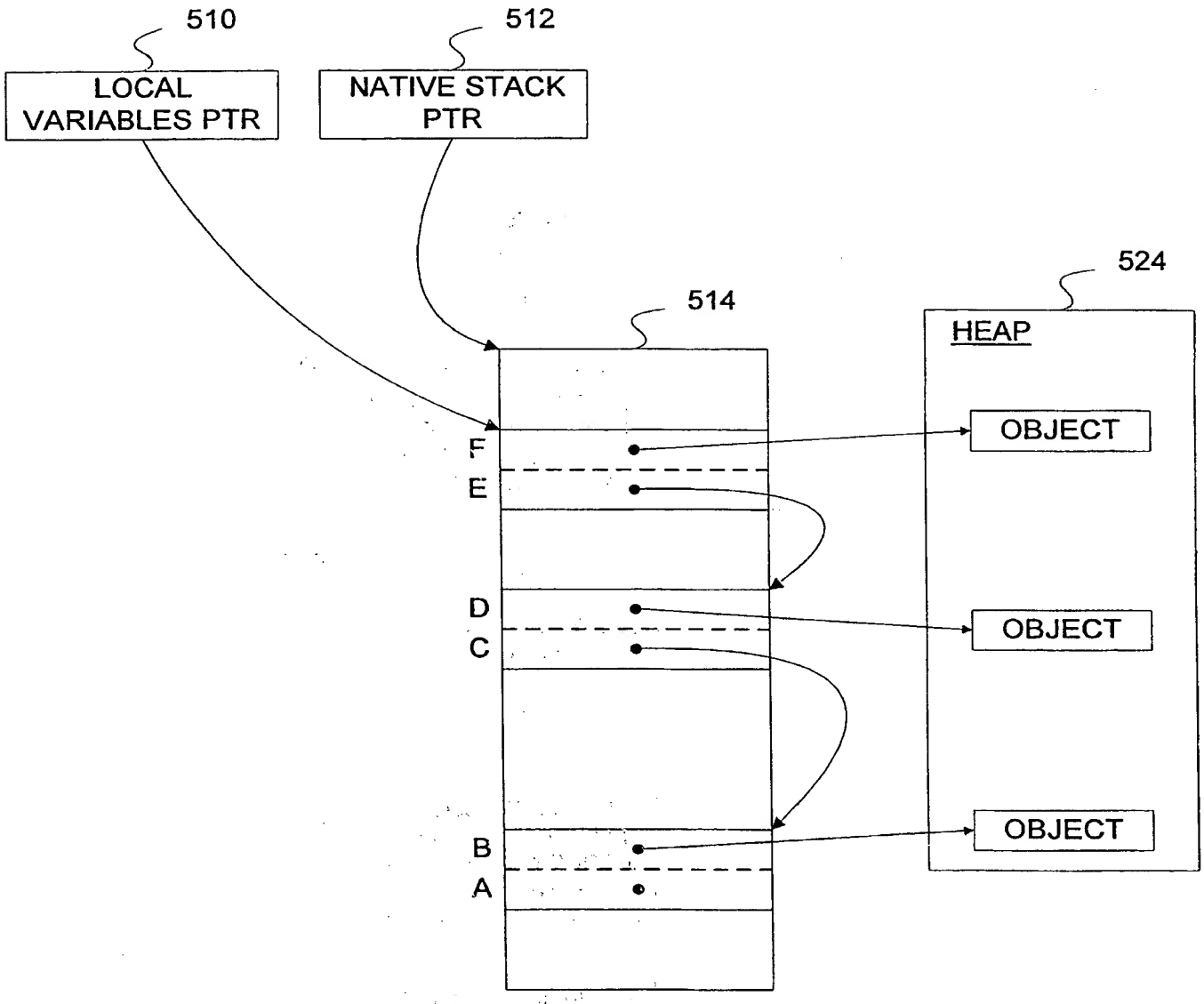


FIG. 5

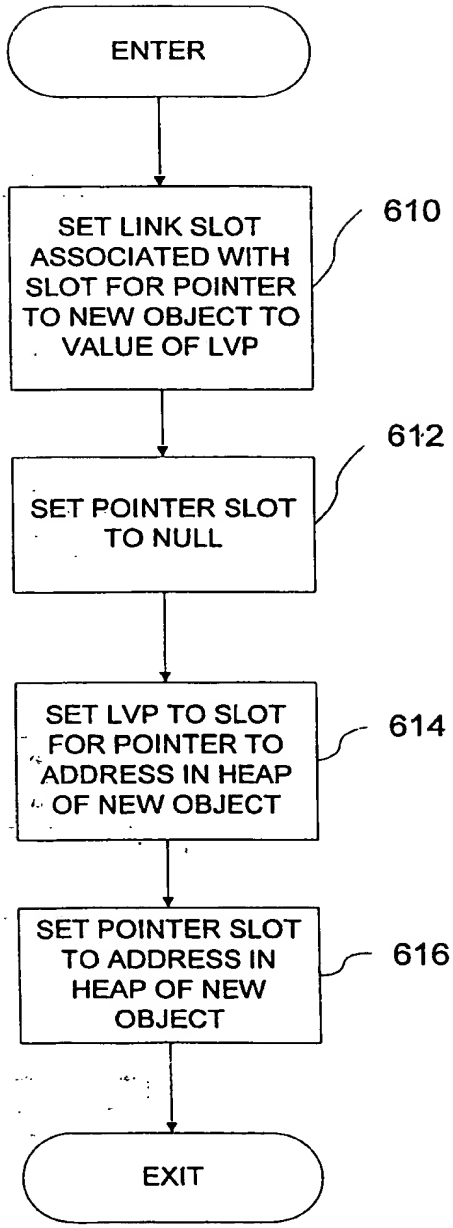


FIG. 6

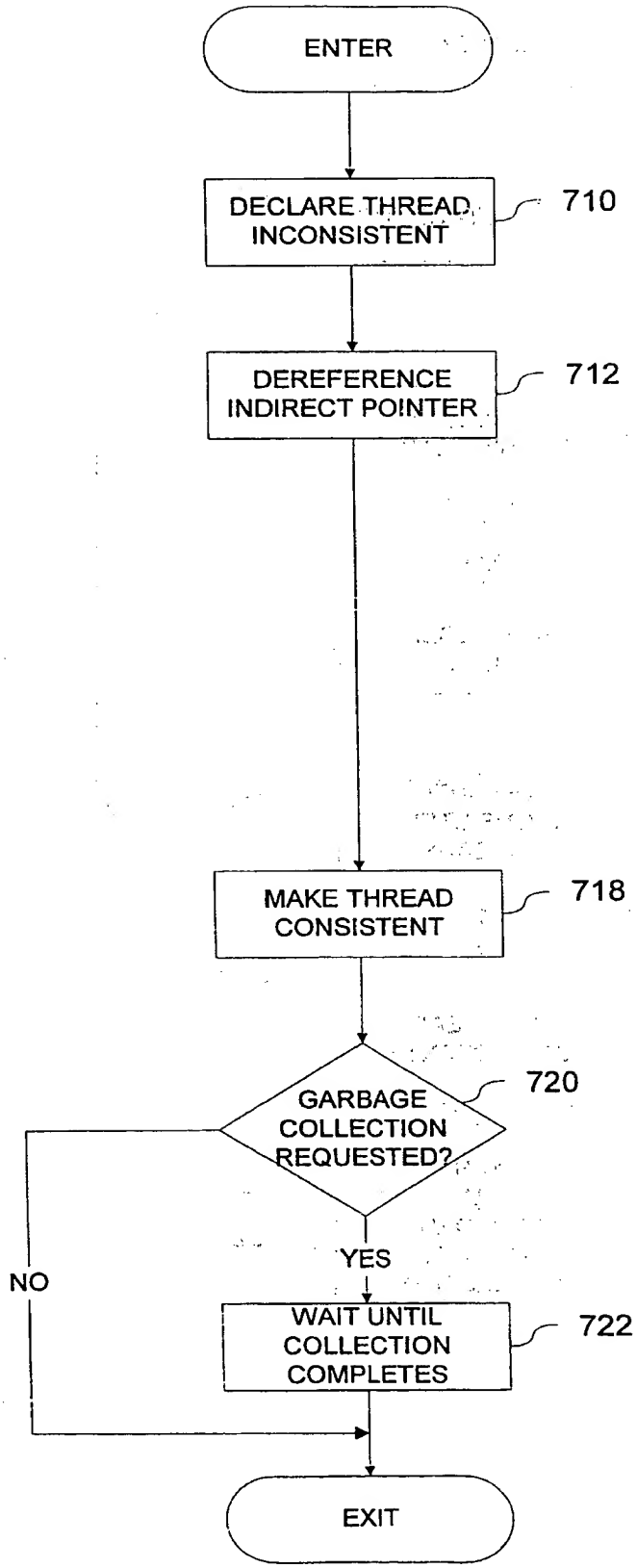


FIG. 7

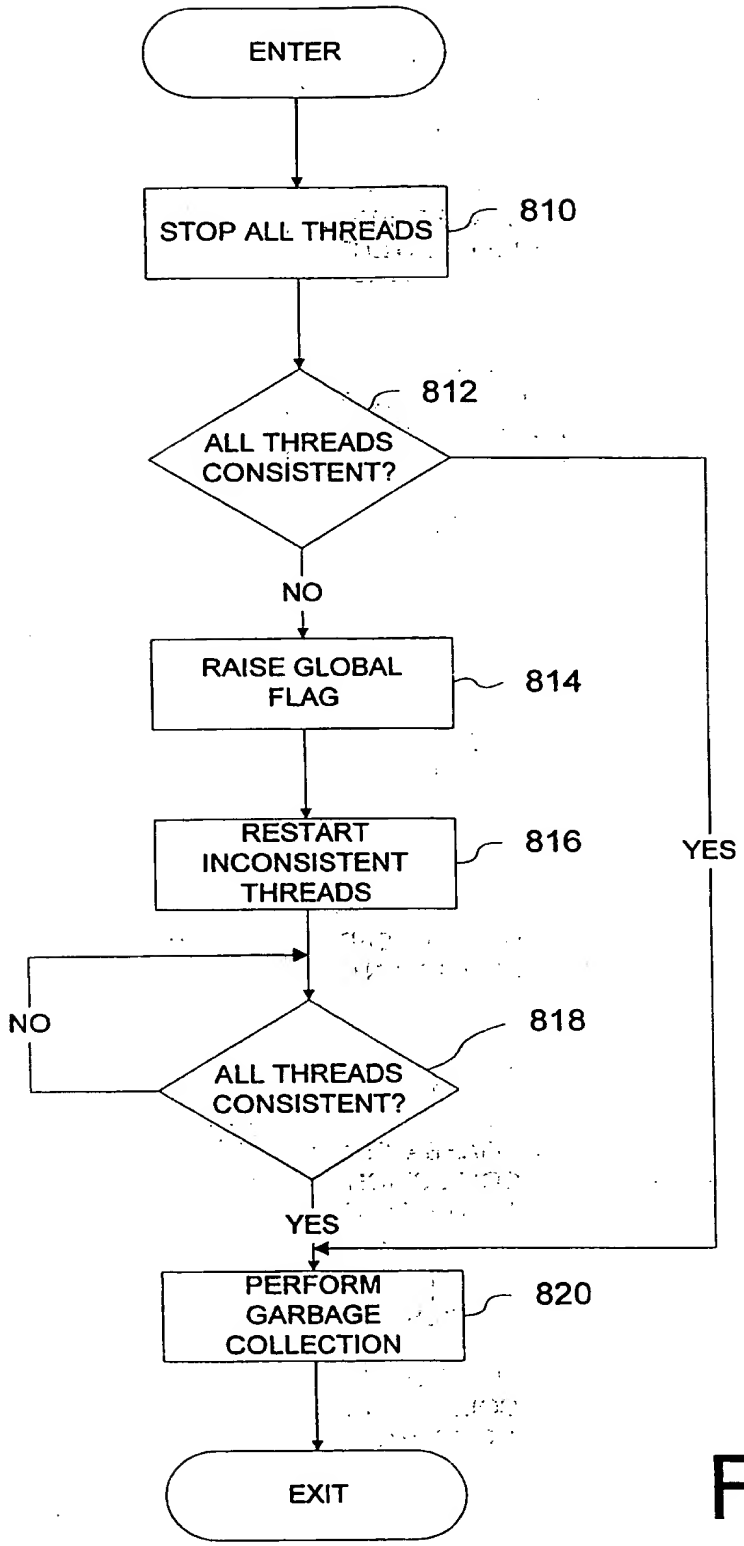


FIG. 8

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 99/18321

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F12/02

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	AGESEN O ET AL: "GARBAGE COLLECTION AND LOCAL VARIABLE TYPE-PRECISION AND LIVENESS IN JAVA TM VIRTUAL MACHINES" ACM SIGPLAN NOTICES,US,ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 33, no. 5, May 1998 (1998-05), page 269-279 XP000766276 ISSN: 0362-1340 cited in the application page 271, left-hand column, line 47 -page 272, left-hand column, line 4 —— -/—	1-52

[X] Further documents are listed in the continuation of box C.

[ ] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
17 December 1999	11/01/2000

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Nielsen, O

1

Form PCT/I A/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Im tional Application No
PCT/US 99/18321

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	DIWAN A ET AL: "COMPILER SUPPORT FOR GARBAGE COLLECTION IN A STATICALLY TYPED LANGUAGE*" ACM SIGPLAN NOTICES,US,ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 27, no. 7, July 1992 (1992-07), page 273-282 XP000332447 ISSN: 0362-1340 page 278, right-hand column, line 2 - line 33	1,6-11, 23-28, 40-45,52
A	KAZUHIRO OGATA: "THE DESIGN AND IMPLEMENTATION OF HOME" ACM SIGPLAN NOTICES,US,ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 27, no. 7, page 44-54 XP000332428 ISSN: 0362-1340 page 47, left-hand column, line 16 -right-hand column, line 38	1,6-11, 23-28, 40-45,52

Form PCT/ISA/210 (continuation of second sheet) (July 1992)